INT3075 Programming and Problem Solving for Mathematics

More on Files and Exceptions

What we already know

- Files are bytes on disk. Two types, text and binary (we are working with text)
- `open` creates a connection between the disk contents and the program
- different modes of opening a file, 'r', 'w', 'a'
- files might have different encodings (default is utf_8)

more of what we know

- all access, reading or writing, to a text file is by the use of strings
- iteration via a `for` loop gathers info from a file opened for reading one line at a time
- we write to a file opened for writing using the `print` function with an argument `file=`

Code Listing

L11-1.py

Review

```python
# Prompt for three values: input file, output file, search string.
# Search for the string in the input file, write results to the
# output file

import sys
def process_file(i_file, o_file, a_str):
    ''' if the a_str is in a line of i_file, add stars
        to the a_str in line, write it out with the
        line number to o_file '''
    line_count_int = 1
    for line_str in i_file:
        if a_str in line_str:
            new_line_str = line_str.replace(a_str, '***'+a_str)
            print('Line {}: {}'.format(line_count_int, new_line_str),\
                  file=o_file)
        line_count_int += 1

try:
    in_file_str = input("File to search:")
    in_file = open(in_file_str, 'r', encoding='utf_8')
except IOError:
    print('{} is a bad file name'.format(in_file_str))
    sys.exit()

out_file_str = input("File to write results to:")
out_file = open(out_file_str, 'w')
search_str = input("Search for what string:")
process_file(in_file, out_file, search_str)
in_file.close()
out_file.close()
```

results, searching for `"This"`

inFile.txt	outFile.txt
This is a test	Line 1: ***This is a test
This is only a test	
Do not pass go	Line 2: ***This is only a test
Do not collect $200	

More ways to read

- `my_file.read()`
  - Reads the entire contents of the file as a string and returns it. It can take an optional argument integer to limit the read to N bytes, that is my_file.read(N)
- `my_file.readline()`

  - Delivers the next line as a string.
- `my_file.readlines()` # note plural
  - Returns a *single list* of all the lines from the file

example file

We'll work with a file called `temp.txt` which has the following file contents

```
First Line
Second Line
Third Line
Fourth Line
```

```
>>> temp_file = open("temp.txt","r")      # open file for reading
>>> first_line_str = temp_file.readline() # read exactly one line
>>> first_line_str
'First line\n'
>>> for line_str in temp_file:       # read remaining lines
        print(line_str)

Second line

Third line

Fourth line

>>> temp_file.readline()                  # file read, return empty str
''
>>> temp_file.close()
```

```
>>> temp_file = open("temp.txt","r")    # open file for reading
>>> temp_file.read(1)                    # read 1 char
'F'
>>> temp_file.read(2)                    # read the next 2 chars
'ir'
>>> temp_file.read()                     # read remaining file
'st line\nSecond line\nThird line\nFourth line\n'
>>> temp_file.read(1)                    # file read, return empty string
''
>>> temp_file.close()
```

```
>>> temp_file = open("temp.txt","r")        # open file for reading
>>> file_contents_list = temp_file.readlines() # read all file lines into a list
>>> file_contents_list
['First line\n', 'Second line\n', 'Third line\n', 'Fourth line\n']
>>>
```

More ways to write

- Once opened, you can write to a file (if the mode is appropriate):
- `my_file.write(s)`
  - Writes the string s to the file
- `my_file.writelines(lst)`
  - write a ***list of strings*** (one at a time) to the file

```python
>>> word_list = ['First', 'Second', 'Third', 'Fourth']
>>> out_file = open('outFile.txt', 'w')
>>> for word in word_list:
...     out_file.write(word + ' line\n')
...
>>> out_file.close()
>>>
```

Universal New Line

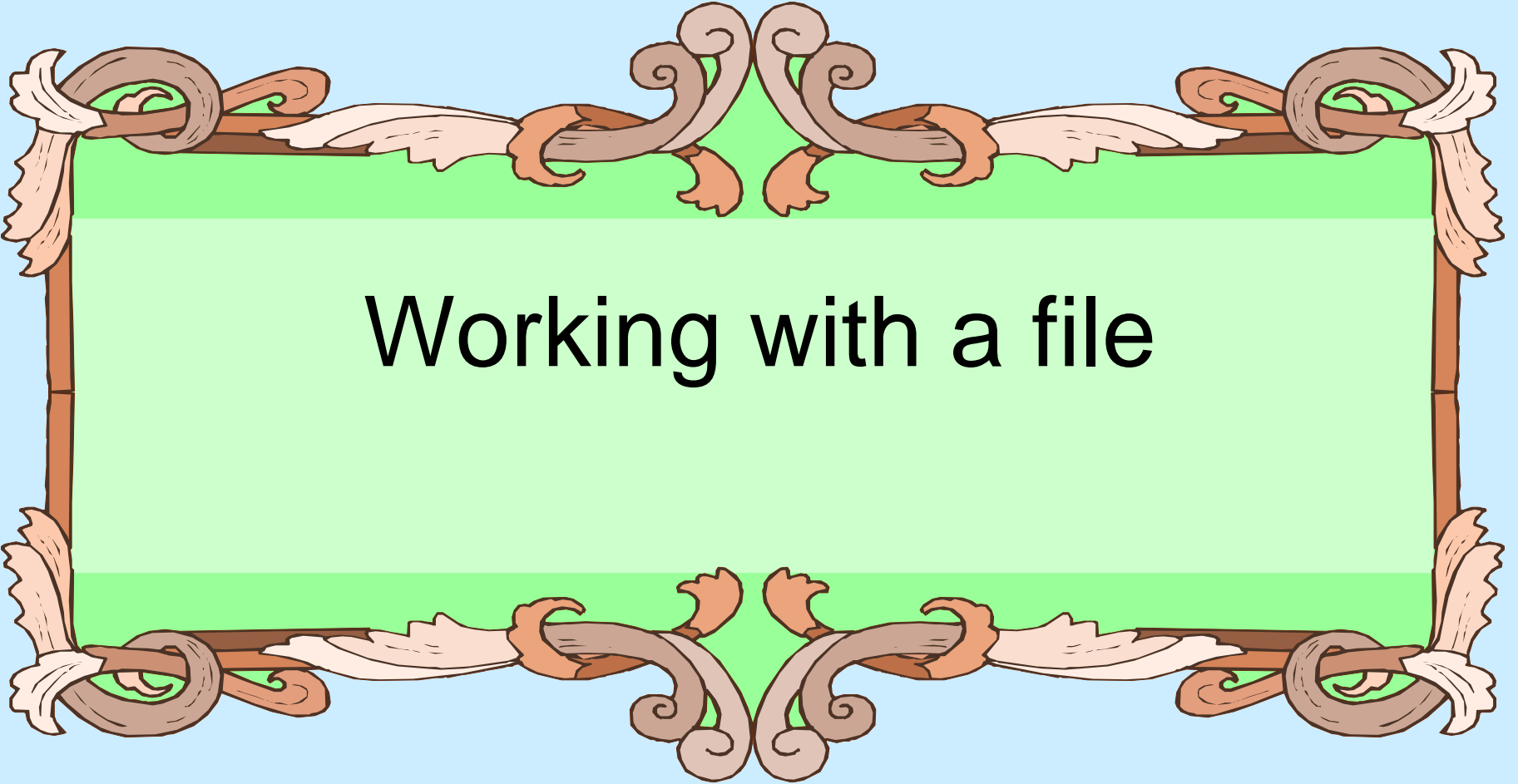# Different OS's, different format

- Each operating system (Windows, OS X, Linux) developed certain standards for representing text

- In particular, they chose different ways to represent the end of a file, the end of a line, etc.

Operating System	Character Combination
Unix & Mac OS X	'\n'
MS Windows	'\r\n'
Mac (pre-OS X)	'\r'

TABLE 14.1 End-of-Line Characters

Universal new line

- To get around this, Python provides ***by default*** a special file option to deal with variations of OS text encoding called Universal new line

- you can over-ride this with an option to `open` called `newline=`
  - look at the docs for what this entails.

Working with a file

The current file position

- Every file maintains a ***current file position***.

- It is the current position in the file, and indicates what the file will read next

- Is set by the mode table above

Remember the file object buffer

- When the disk file is opened, the contents of the file are copied into the buffer of the file object

- Think of the file object as a very big list, where every index is one of the pieces of information of the file

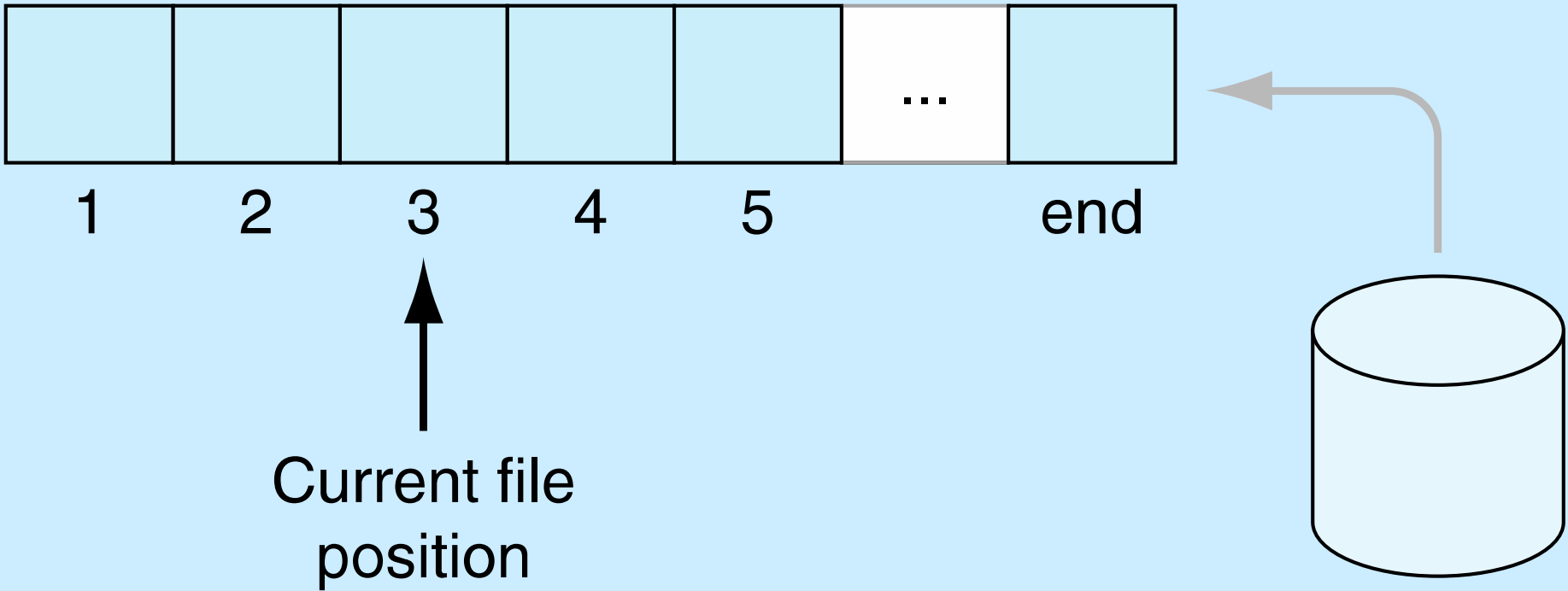- The current position is the present index in that list

File object buffer

1   2   3   4   5   end

Current file position

FIGURE 14.1 Current file position.

the tell() method

- The `tell()` method tells you the current file position

- The positions are in bytes (think characters for UTF-8) from the beginning of the file

my_file.tell() => 42L

the seek() method

- the `seek()` method updates the current file position to a new file index (in bytes offset from the beginning of the file)
- fd.seek(0)   # to the beginning of the file
- fd.seek(100) # 100 bytes from beginning

counting bytes is a pain

- counting bytes is a pain
- `seek` has an optional argument set:
  - 0: count from the beginning
  - 1: count for the current file position
  - 2: count from the end (backwards)

every read moves current forward

- every read/readline/readlines moves the current pos forward

- when you hit the end, every read will just yield `''` (empty string), since you are at the end

  – no indication of end-of-file this way!

- you need to seek to the beginning to start again (or close and open, seek is easier)

```
>>> test_file = open('temp.txt','r')
>>> test_file.tell()                # where is the current file position?
0
>>> test_file.readline()            # read first line
'First Line\n'
>>> test_file.tell()                # where are we now?
11
>>> test_file.seek(0)               # go to beginning
0
>>> test_file.readline()            # read first line again
'First Line\n'
>>> test_file.readline()            # read second line
'Second Line\n'
>>> test_file.tell()                # where are we now?
23
>>> test_file.seek(0,2)             # go to end
46
>>> test_file.tell()                # where are we now?
46
>>> test_file.readline()            # try readline at end of file: nothing there
''
>>> test_file.seek(11)              # go to the end of the first line (see tell above)
11
>>> test_file.readline()            # when we read now we get the second line
'Second Line\n'
>>> test_file.close()
>>> test_file.readline()            # Error: reading after file is closed
Traceback (most recent call last):
  File "<pyshell#65>", line 1, in <module>
    test_file.readline()
ValueError: I/O operation on closed file.
>>>
```

`with` statement

`open` and `close` occur in pairs (or should) so Python provides a shortcut, the `with` statement

- creates a context that includes an exit which is invoked automatically
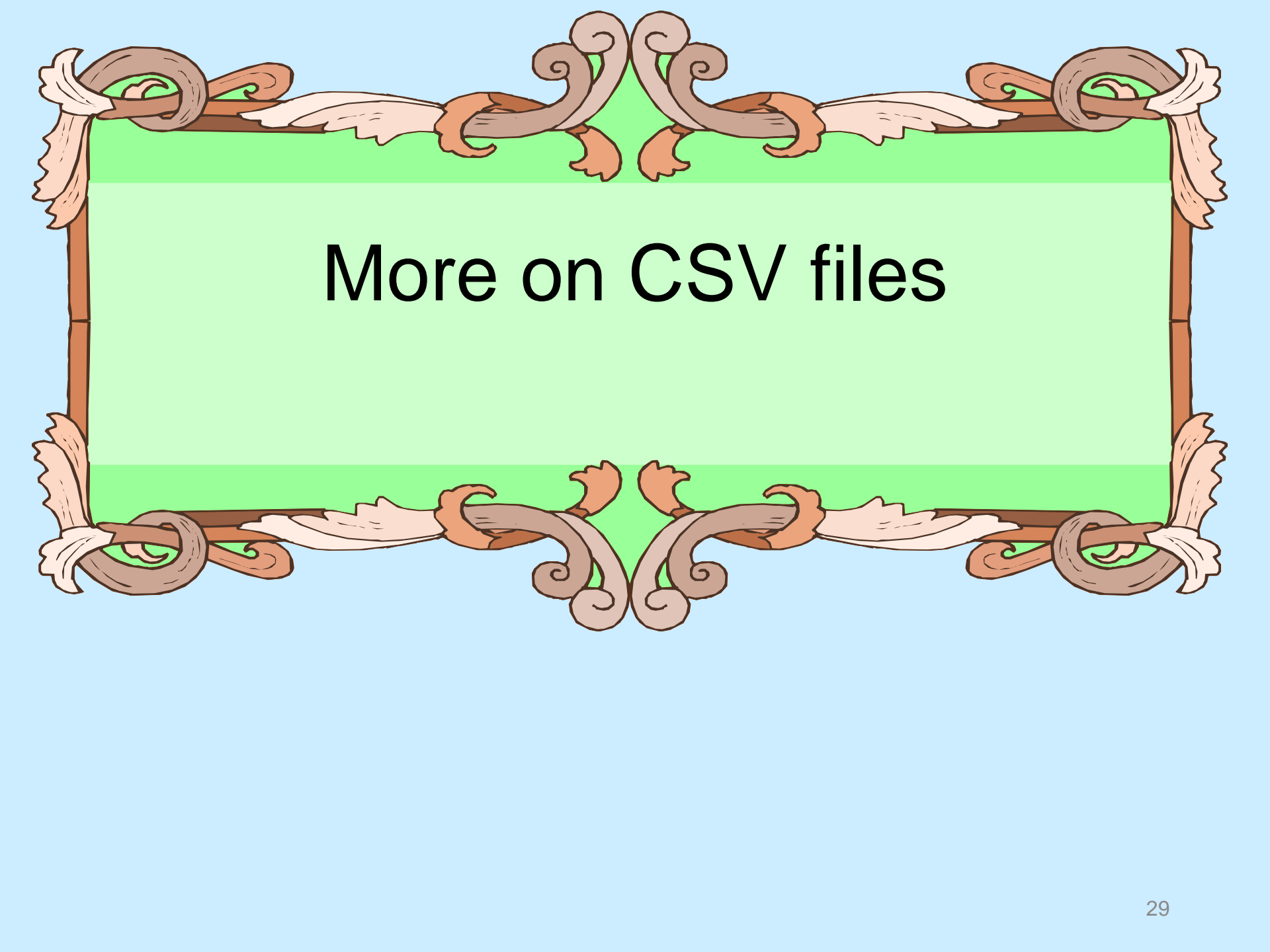- for files, the exit is to close the file

```
with expression as variable:
        suite
```

File is closed automatically when the suite ends

```
>>> with open('temp.txt') as temp_file:
...     temp_file.readlines()
...
['First line\n', 'Second line\n', 'Third line\n', 'Fourth line\n']
>>>
```

read(size=1)

- you can use the `read()` method to read just one byte at a time, and in combination with `seek` move around the file and "look for things". Once current is set, you can begin reading again

More on CSV files

spreadsheets

- The spreadsheet is a very popular, and powerful, application for manipulating data
- Its popularity means there are many companies that provide their own version of the spreadsheet
- It would be nice if those different versions could share their data

CSV, basic sharing

- A basic approach to share data is the comma separated value (CSV) format
    - it is a text format, accessible to all apps
    - each line (even if blank) is a row
    - in each row, each value is separated from the others by a comma (even if it is blank)
    - cannot capture complex things like formula

Spread sheet and corresponding CSV file

Name	Exam1	Exam2	Final Exam	Overall Grade
Bill	75.00	100.00	50.00	**75.00**
Fred	50.00	50.00	50.00	**50.00**
Irving	0.00	0.00	0.00	**0.00**
Monty	100.00	100.00	100.00	**100.00**
Average				**56.25**

FIGURE 14.2 A simple spreadsheet from Microsoft Excel 2008.

```
Name,Exam1,Exam2,Final Exam,Overall Grade
Bill,75.00,100.00,50.00,75.00
Fred,50.00,50.00,50.00,50.00
Irving,0.00,0.00,0.00,0.00
Monty,100.00,100.00,100.00,100.00

Average,,,,56.25
```

Even CSV isn't universal

- As simple as that sounds, even CSV format is not completely universal
  - different apps have small variations
- Python provides a module to deal with these variations called the csv module
- This module allows you to read spreadsheet info into your program

csv reader

- import the csv module

- open the file as normally, creating a file object.

- create an instance of a csv reader, used to iterate through the file just opened
  - you provide the file object as an argument to the constructor

- iterating with the reader object yields a row as a list of strings

Code Listing

L11-2.py

(and output)

```python
import csv
workbook_file = open('Workbook1.csv','r')
workbook_reader = csv.reader(workbook_file)

for row in workbook_reader:
    print(row)


workbook_file.close()
```

```
>>>
['Name', 'Exam1', 'Exam2', 'Final Exam', 'Overall Grade']
['Bill', '75.00', '100.00', '50.00', '75.00']
['Fred', '50.00', '50.00', '50.00', '50.00']
['Irving', '0.00', '0.00', '0.00', '0.00']
['Monty', '100.00', '100.00', '100.00', '100.00']
[]
['Average', '', '', '', '56.25']
>>>
```
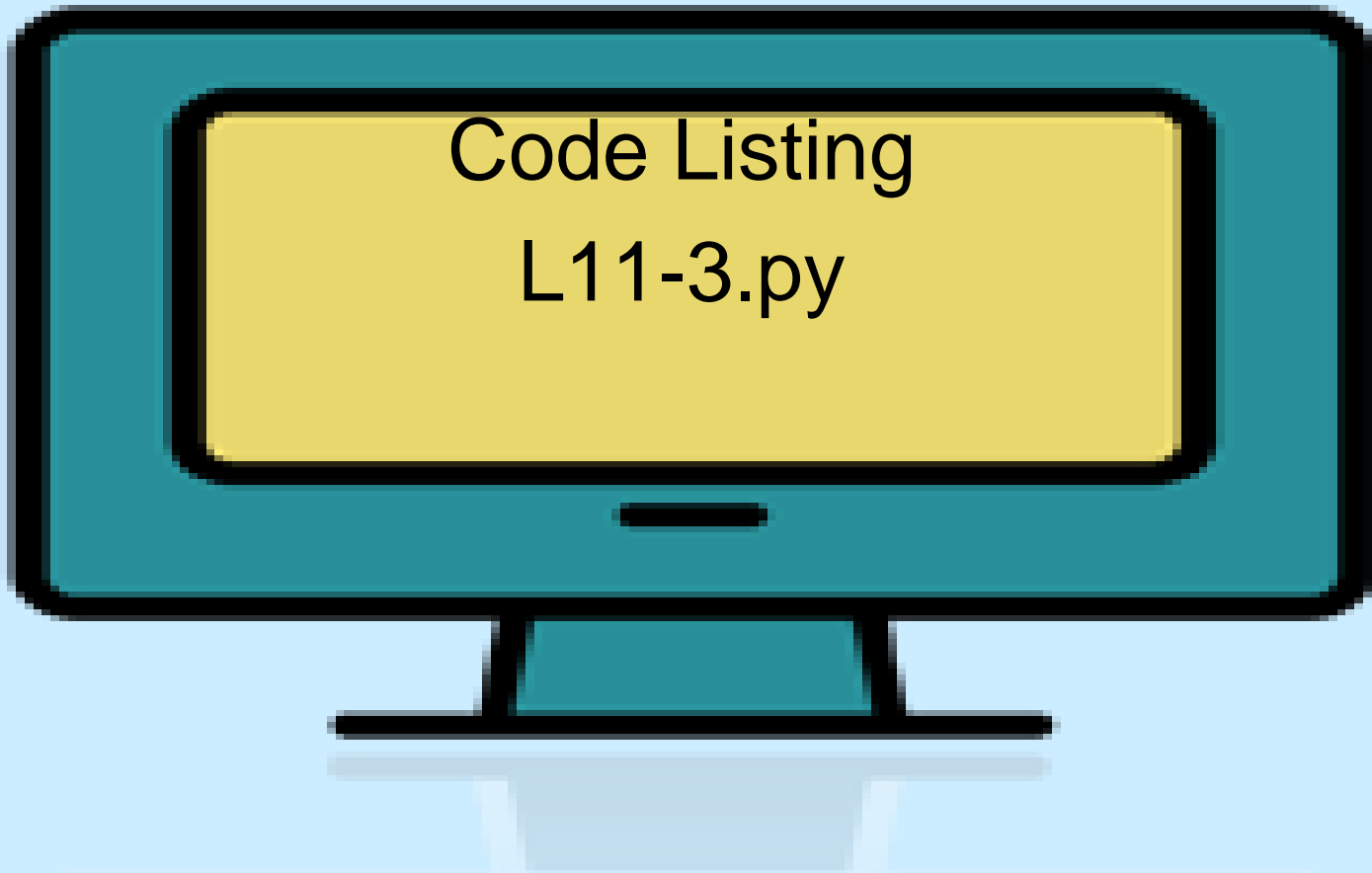
things to note

- Universal new line is working by default
  - needed for this worksheet
- A blank line in the CSV shows up as an empty list
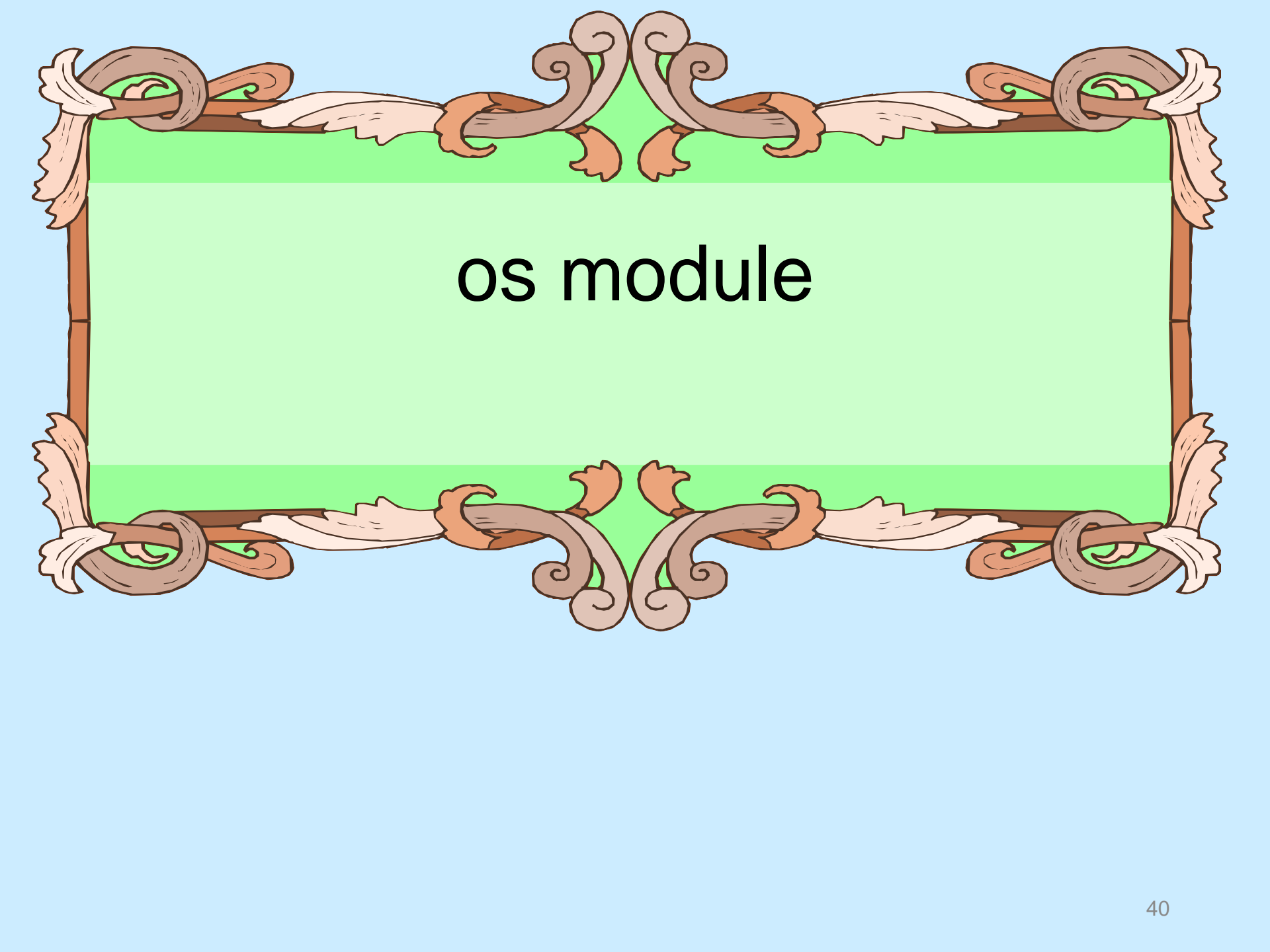- empty column shows up as an empty string in the list

csv writer

much the same, except:

- the opened file must have write enabled
- the method is writerow, and it takes a ***list of strings*** to be written as a row

Code Listing

L11-3.py

This code listing is a good example of reading, modifying and then writing out a CSV file that could be read by a spreadsheet

os module

What is the os module

- The `os` module in Python is an interface between the operating system and the Python language.

- As such, it has many sub-functionalities dealing with various aspects.

- We will look mostly at the file related stuff

What is a directory/folder?

- Whether in Windows, Linux or on OS X, all OS's maintain a ***directory structure***.

- A directory is a container of files or other directories

- These directories are arranged in a hierarchy or tree

Tree

- it has a ***root*** node, with ***branch*** nodes, ends in ***leaf*** nodes

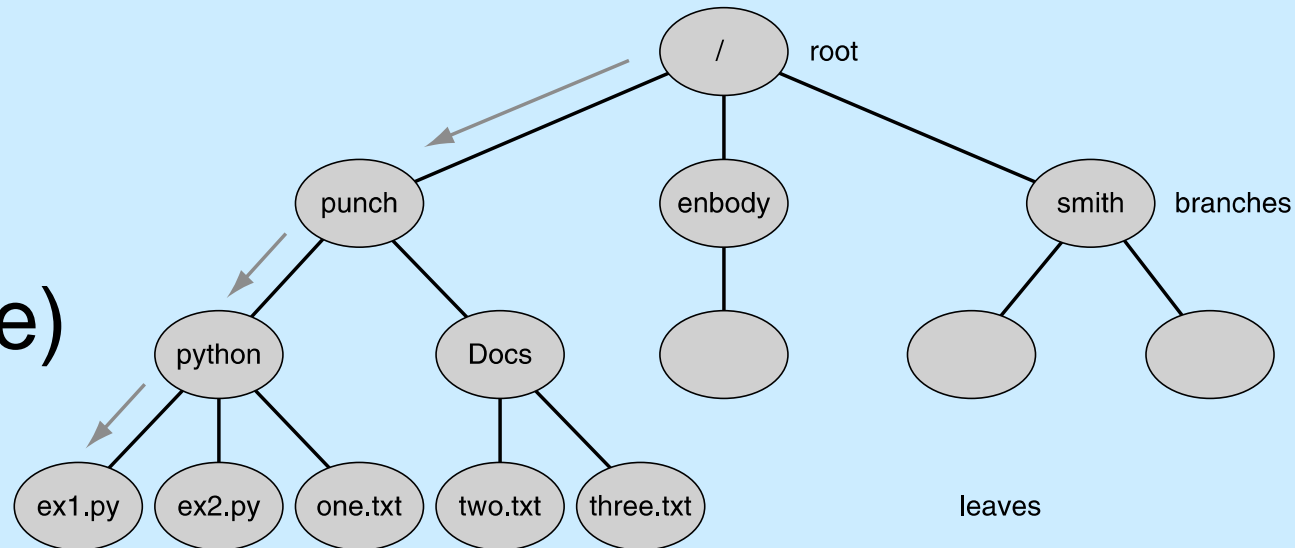- the directory structure is hierarchy (tree)

FIGURE 14.4 Directory tree with path /punch/python/ex1.py, marked with arrows.

43

Directory tree

- Directories can be organized in a hierarchy, with the root directory and subsequent branch and leaf directories
- Each directory can hold files or other directories
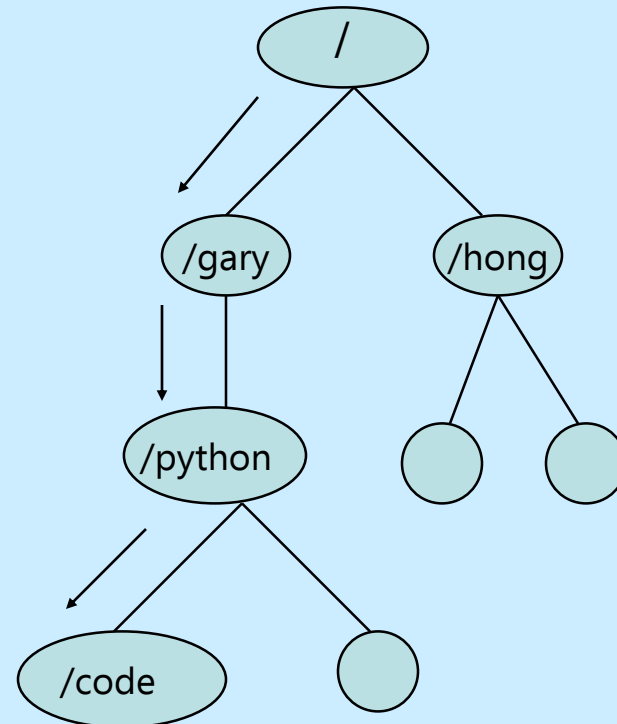
file path is a path through the tree

A path to a file is a path through the hierarchy to the node that contains a file

```
/gary/python/code/myCode.py
```

– path is from the root node /, to the bill directory, to the python directory, to the code directory where the file myCode.py resides

the / in a path

- think of / as an operator, showing something is a directory
- follow the path, the leaf is either a directory or file

a path String

- a valid path string for python is a string which indicates a valid path in the directory structure
- Thus `'/Users/gary/python/code.py'` is a valid path string

different 'paths' for different os

- It turns out that each OS has its own way of specifying a path
  - C:\gary\python\myFile.py
  - /Users/gary/python/myFile.py
- Nicely, Python knows that and translates to the appropriate OS

Two special directory names

- The directory name `'.'` is shortcut for the name of the current directory you are in as you traverse the directory tree
- The directory name `'..'` is a shortcut for the name of the parent directory of the current directory you are in

Some os commands

- `os.getcwd()` Returns the full path of the current working directory

- `os.chdir(path_str)` Change the current directory to the path provided

- `os.listdir(path_str)` Return a list of the files and directories in the path (including `'.'`)

```
>>> import os                        # load the os package
>>> os.chdir("/punch/python")        # change to the example starting point
>>> os.getcwd()                      # check that we are there
'/punch/python'
>>> os.listdir(".")  # list contents of current directory, indicated by "."
['ex1.py', 'ex2.py', 'one.txt']
>>> dir_list = os.listdir(".")       # we can give that list a name
>>> dir_list
['ex1.py', 'ex2.py', 'one.txt']
>>> os.listdir("/punch")             # list the contents at some path
['Docs', 'python']
```

Some more os commands

- `os.rename(source_path_str, dest_path_str)` Renames a file or directory

- `os.mkdir(path_str)` make a new directory. So `os.mkdir('/Users/bill/python/new')` creates the directory `new` under the directory `python`.

- `os.remove(path_str)` Removes the file

- `os.rmdir(path_str)` Removes the directory, but the directory must be empty
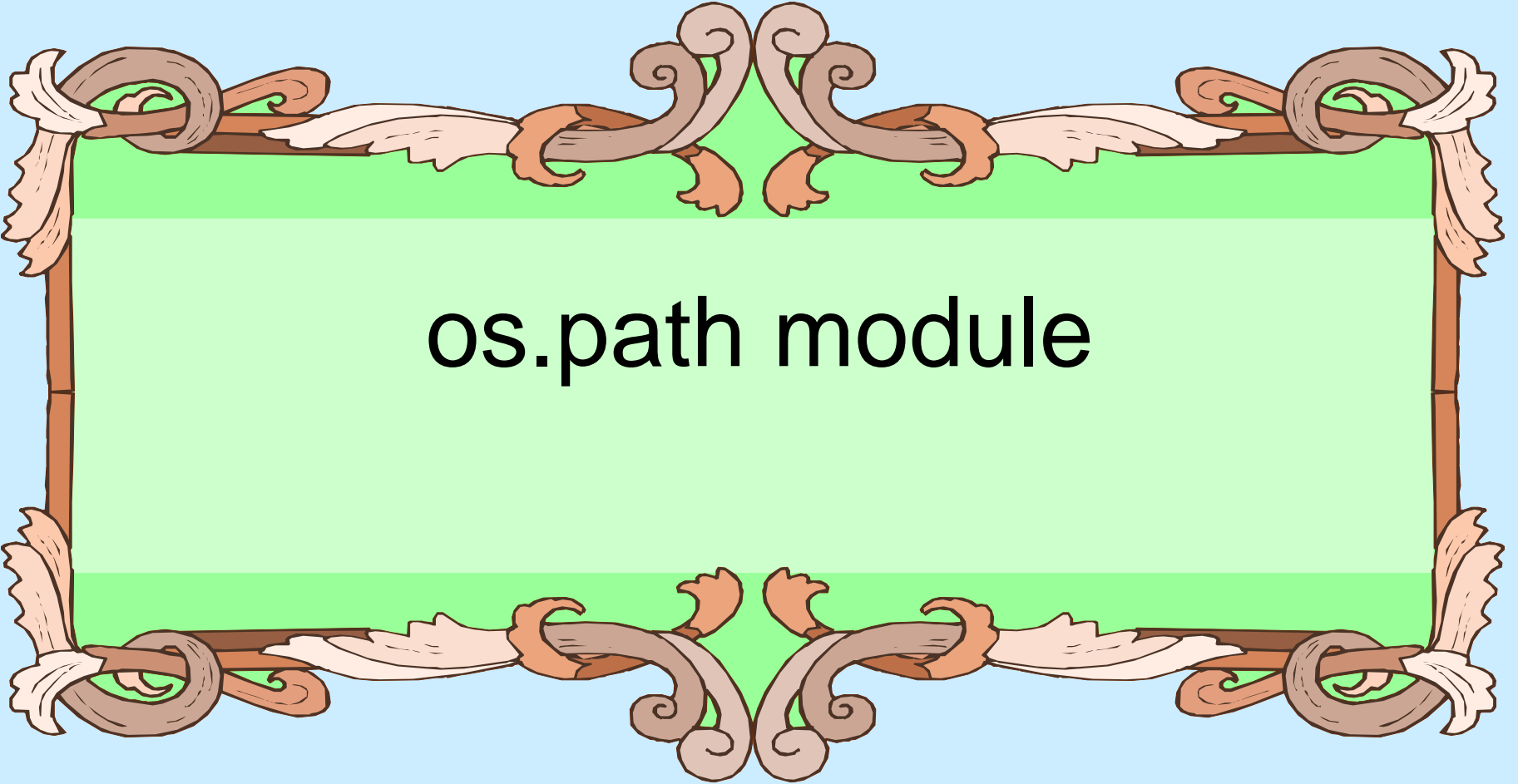
the walk function

- `os.walk(path_str)` Starts at the directory in `path_str`. It yields three values:
  - `dir_name`, name of the current directory
  - `dir_list`, list of subdirectories in the directory
  - `files`, list of files in the directory
- If you iterate through, walk will visit every directory in the tree. Default is top down

Walk example

```
>>> os.getcwd()                                # check our starting point
'/punch'
>>> for dir_name, dirs, files in os.walk("."):  # "walk" in the current directory
        print(dir_name, dirs, files)

. ['Docs', 'python'] []   # current directory, list of 2 subdirectories, no files
./Docs [] ['three.txt', 'two.txt']    # Does directory, no subdirectories, 3 files
./python [] ['ex1.py', 'ex2.py', 'one.txt'] # directory, no subdirectories, 3 files
>>>
```

os.path module

os.path module

allows you to gather some info on a path's existence

- `os.path.isfile(path_str)` is this a path to an existing file (T/F)

- `os.path.isdir(path_str)` is this a path to an existing directory (T/F)

- `os.path.exists(path_str)` the path (either as a file or directory) exists (T/F)

os.path names

assume `p = '/Users/bill/python/myFile.py'`

- `os.path.basename(p)` returns `'myFile.py'`
- `os.path.dirname(p)` returns `'/Users/bill/python'`
- `os.path.split(p)` returns `['Users/bill/python','myFile.py']`
- `os.path.splitext(p)` returns `'/Users/bill/python/myFile', '.py'`
- `os.path.join(os.path.split(p)[0],'other.py')` returns `'/Users/bill/python/other.py'`

Code Listing
L11-4.py

Utility to find strings in files

- The main point of this function is to look through all the files in a directory structure and see if a particular string exists in any of those files

- Pretty useful for mining a set of files

- lots of comments so you can follow

```python
def check(search_str,count,files_found_list,dirs_found_list):
    for dirname,dir_list,file_list in os.walk("."):     # walk the subtree
        for f in file_list:
            if os.path.splitext(f)[1] == ".txt":    # if it is a text file
                count = count + 1                    # add to count of files examined
                a_file = open(os.path.join(dirname,f),'r')  # open text file
                file_str = a_file.read()             # read whole file into string

                if search_str in file_str:           # is search_str in file?
                    filename = os.path.join(dirname,f)   # if so, create path
                                                          # for file
                    files_found_list.append(filename)     # and add to file list
                    if dirname not in dirs_found_list:    # if directory is not
                        dirs_found_list.append(dirname)   # and directory list
                a_file.close()
    return count
```

More Exceptions

What we already know

try-except suite to catch errors:

```
try:
    suite to watch
except ParticularError
    error suite
```

more of what we know

- `try` suite contains code that we want to watch:

  - if an error occurs the try suite stops and looks for an except suite that can handle the error

- `except` suite has a particular error it can handle and a suite of code for handling that error

Error Flow

1. Error occurs here.

Try:
    statement
    statement
    statement

2. Check for correct type of exception.

except PythonException1:
    statement
    statement

3. Execute the exception block.

except PythonException2:
    statement
    statement

4. Skip any more exception blocks.

statement
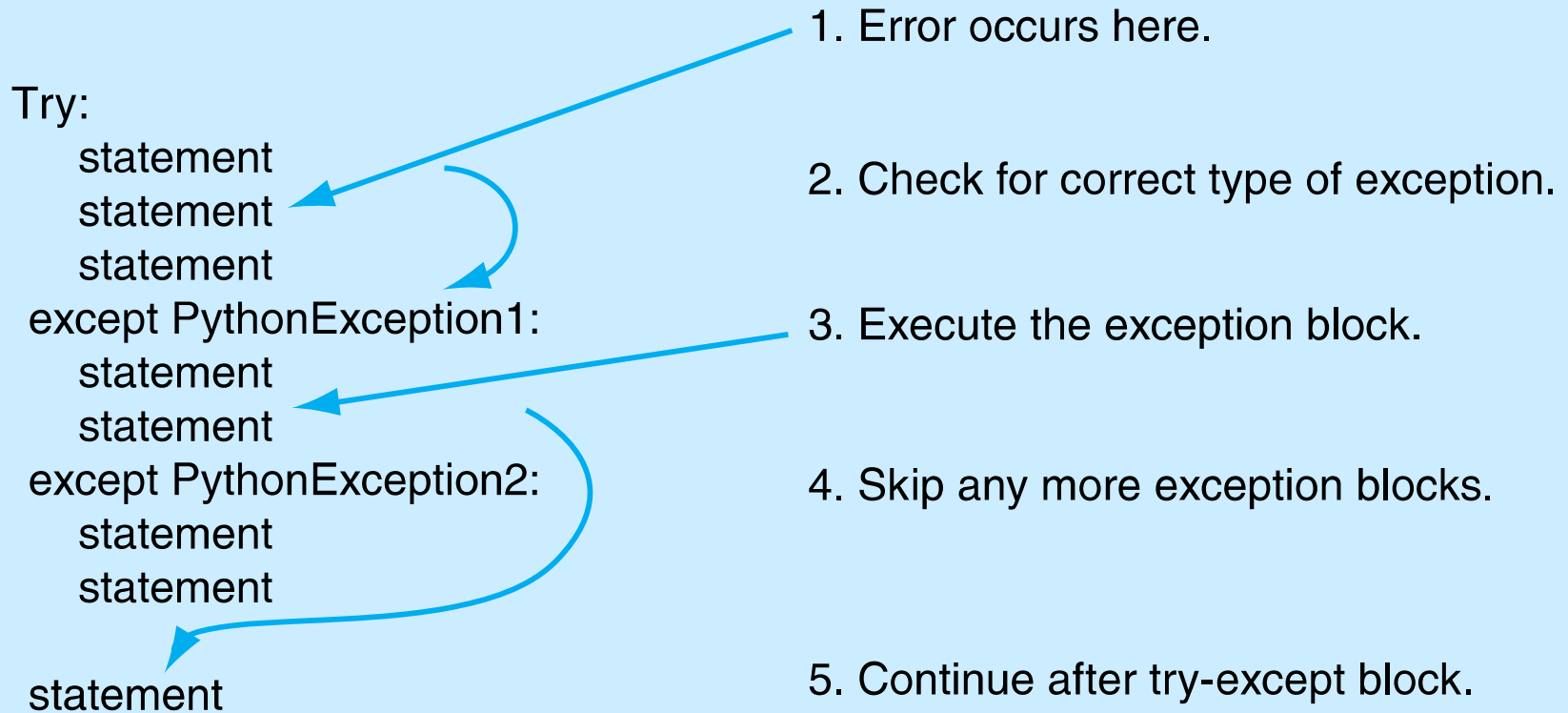
5. Continue after try-except block.

FIGURE 14.5 Exception flow.

Code Listing
L11-5.py

```
1 try:
2     print("Entering the try suite")
3     dividend = float(input("Provide a dividend to divide:"))
4     divisor = float(input("Provide a divisor to divide by:"))
5     result = dividend/divisor
6     print("{:2.2f} divided by {:2.2f} yields {:2.2f}".\
7         format(dividend,divisor,result))
8 except ZeroDivisionError:
9     print("Divide by 0 error")
10 except ValueError:
11     print("Value error, could not convert to a float")
12
13 print("Continuing on with the rest of the program")
```

```
>>>
Entering the try suite
Provide a dividend to divide:10
Provide a divisor to divide by:2
10.00 divided by 2.00 yields 5.00
Continuing on with the rest of the program
>>> ============================== RESTART ==============================
>>>
Entering the try suite
Provide a dividend to divide:10
Provide a divisor to divide by:a
Value error, could not convert to a float
Continuing on with the rest of the program
```

Continuing …

```
>>> =============================== RESTART ===============================
>>>
Entering the try suite
Provide a dividend to divide:10
Provide a divisor to divide by:0
Divide by 0 error
Continuing on with the rest of the program
>>> =============================== RESTART ===============================
>>>
Entering the try suite
Provide a dividend to divide:

Traceback (most recent call last):
  File "/Users/bill/book/v3.5/chapterExceptions/divide.py", line 3, in <module>
    dividend = float(input("Provide a dividend to divide:"))
KeyboardInterrupt
>>>
```

Check for specific exceptions

- Turns out that you don't have to check for an exception type. You can just have an exception without a particular error and it will catch anything

- That is a bad idea. How can you fix (or recover from) an error if you don't know the kind of exception

- Label your exceptions, all that you expect!

What exceptions are there?

- In the present Python, there is a set of exceptions that are pre-labeled.

- To find the exception for a case you are interested it, easy enough to try it in the interpreter and see what comes up

- The interpreter tells you what the exception is for that case.

```
BaseException
    +– SystemExit
    +– KeyboardInterrupt
    +– GeneratorExit
    +– Exception
        +– StopIteration              +– RuntimeError
        +– ArithmeticError            |   +– NotImplementedError
        |   +– FloatingPointError     +– SyntaxError
        |   +– OverflowError          |   +– IndentationError
        |   +– ZeroDivisionError      |       +– TabError
        +– AssertionError             +– SystemError
        +– AttributeError             +– TypeError
        +– BufferError                +– ValueError
        +– EnvironmentError           |   +– UnicodeError
        |   +– IOError                |       +– UnicodeDecodeError
        |   +– OSError                |       +– UnicodeEncodeError
        |       +– WindowsError       |       +– UnicodeTranslateError
        |       +– VMSError           +– Warning
        +– EOFError                       +– DeprecationWarning
        +– ImportError                    +– PendingDeprecationWarning
        +– LookupError                    +– RuntimeWarning
        |   +– IndexError                 +– SyntaxWarning
        |   +– KeyError                   +– UserWarning
        +– MemoryError                    +– FutureWarning
        +– NameError                      +– ImportWarning
        |   +– UnboundLocalError          +– UnicodeWarning
        +– ReferenceError                 +– BytesWarning
                                          +– ResourceWarning
```

FIGURE 14.6 Python exceptions.

Examples

```
>>> 100/0
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    100/0
ZeroDivisionError: integer division or modulo by zero


>>> open('badFileName')
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    open('badFileName')
IOError: [Errno 2] No such file or directory: 'badFileName'
```
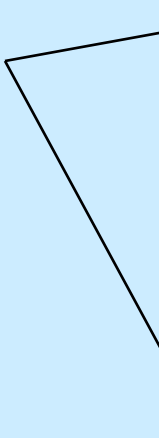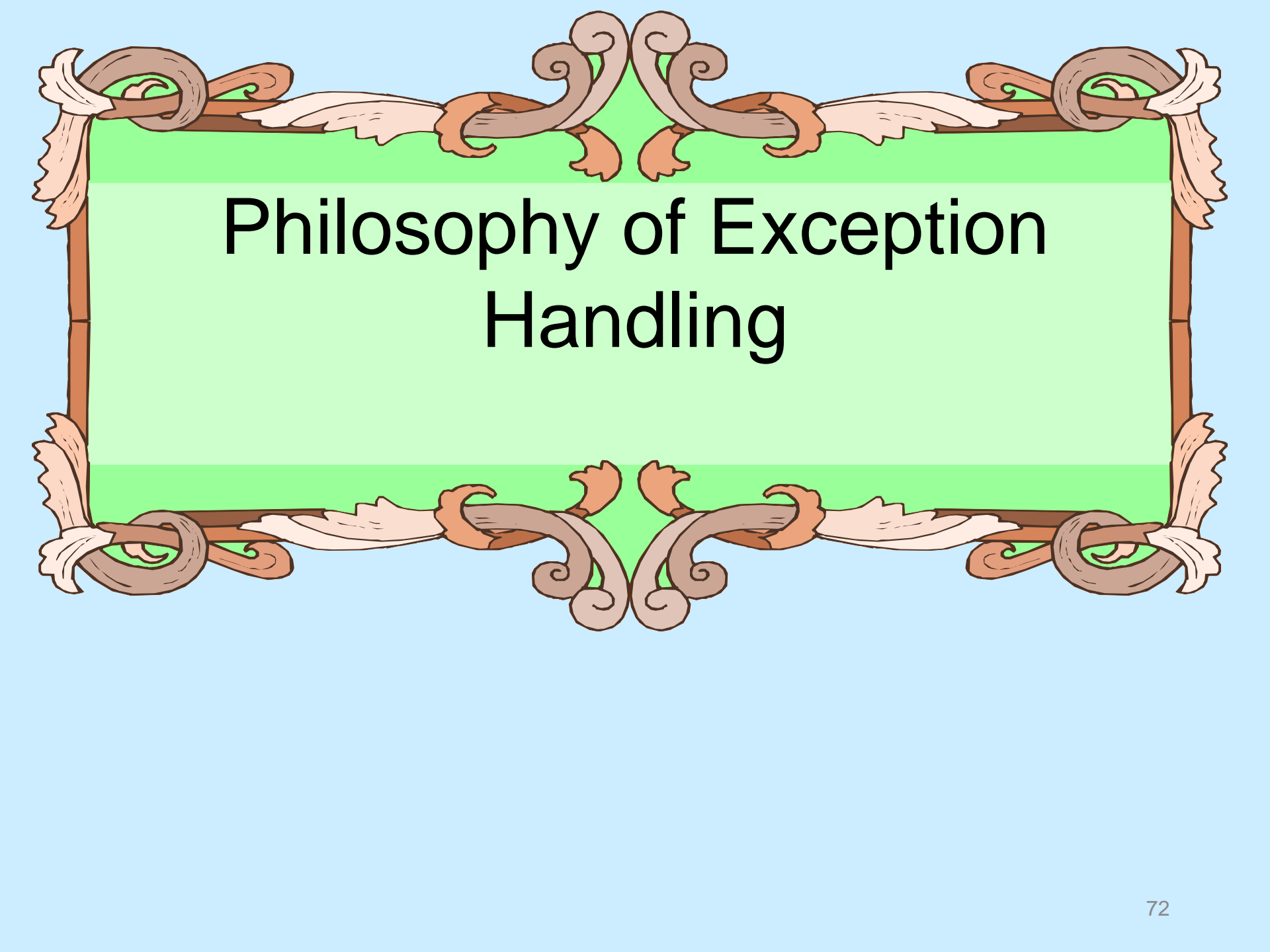
error
names
CAPS
matter!

Philosophy of Exception Handling

Dealing with problems

Two ways to deal with exceptions

- *LBYL*: Look Before you Leap

- *EAFP*: Easier to Ask Forgiveness than Permission (famous quote by Grace Hopper)

Look Before You Leap

- By this we mean that before we execute a statement, we check all aspects to make sure it executes correctly:
  - if it requires a string, check that
  - if it requires a dictionary key, check that
- Tends to make code messy. The heart of the code (what you want it to do) is hidden by all the checking.

Easier to Ask Forgiveness than Permission

- By this we mean, run any statement you want, no checking required

- However, be ready to "clean up any messes" by catching errors that occur

- The `try` suite code reflects what you want to do and the `except` code what you want to do on error. Cleaner separation!

Python likes EAFP

- Code Python programmers support the ***EAFP*** approach:
  - run the code, let the `except` suites deal with the errors. Don't check first.

Code Listing

L11-6.py

```python
# check whether int conversion will raise an error, two examples.
# Python Idioms, http://jaynes.colorado.edu/PythonIdioms.html

#LBYL, test for the problematic conditions
def test_lbyl (a_str):
    if not isinstance(a_str, str) or not a_str.isdigit:
        return None
    elif len(a_str) > 10:       #too many digits for int conversion
        return None
    else:
        return int(a_str)

#EAFP, just try it, clean up any mess with handlers
def test_eafp(a_str):
    try:
        return int(a_str)
    except (TypeError, ValueError, OverflowError): #int conversion failed
        return None
```

Extensions to the basic Exception Model

`finally` suite

- you can add a `finally` suite at the end of the `try`/`except` group

- the finally suite is run as you exit the `try`/`except` suite, ***no matter whether an error occurred or not***.

  - even if an exception raised in the `try` suite was not handled!

- Gives you an opportunity to clean up as you exit the `try`/`except` group

`finally` and `with`

`finally` is related to a `with` statement:

- creates a context (the `try` suite)
- has an exit, namely execute the `finally` suite

else

- One way to think about things is to think of the `try` as a kind of condition (an exception condition) and the `except`s as conditional clauses

- if an exception occurs then you match the exception

- the `else` clause covers the non-exception condition. It runs when the `try` suite does not encounter an error.

The whole thing

```
try:
    code to try
except PythonError1:
    exception code
except PythonError2:
    exception code
except:
    default except code
else:
    non exception case
finally:
    clean up code
```

Code Listing
L11-7.py

```python
# all aspects of exceptions

def process_file(data_file):
    """Print each line of a file with its line number."""
    count = 1
    for line in data_file:
        print('Line ' + str(count) + ': ' + line.strip())
        count = count + 1


while True:      # loop forever: until "break" is encountered
    filename = input('Input a file to open: ')
    try:
        data_file = open(filename)
    except IOError:                          # we get here if file open failed
        print('Bad file name; try again')
    else:
        # no exception so let's process the file
        print('Processing file',filename)
        process_file(data_file)
        break        # exit "while" loop (but do "finally" block first)
    finally:          # we get here whether there was an exception or not
        try:
            data_file.close()
        except NameError:
            print('Going around again')

print('Line after the try-except group')
```

Raising and creating your own exceptions

invoking yourself, `raise`

- You can also choose to invoke the exception system anytime you like with the `raise` command

```
raise MyException
```

- you can check for odd conditions, `raise` them as an error, then catch them

- they must be part of the existing exception hierarchy in Python

Non-local catch

- Interestingly, the `except` suite does not have to be right next to the try suite.

- In fact, the `except` that catches a `try` error can be in another function

- Python maintains a chain of function invocations. If an error occurs in a function and it cannot catch it, it looks to the function that called it to catch it

Make your own exception

- You can make your own exception.
- Exceptions are classes, so you can make a new exception by making a new subclass:

```
class MyException (IOError):
    pass
```

- When you make a new class, you can add your own exceptions.

Code Listing
L11-8.py

Part 1

```python
import string

# define our own exceptions
class NameException (Exception):
    ''' For malformed names'''
    pass
class PasswordException (Exception):
    ''' For bad password '''
    pass
class UserException (Exception):
    ''' Raised for existing or missing user '''
    pass

def check_pass(pass_str, target_str):
    """Return True, if password contains characters from target."""
    for char in pass_str:
        if char in target_str:
            return True
    return False

class PassManager(object):
    """A class to manage a dictionary of passwords with error checking."""
    def __init__(self, init_dict=None):
        if init_dict==None:
            self.pass_dict={}
```

Part 2

```python
26          else:
27              self.pass_dict = init_dict.copy()
28
29      def dump_passwords(self):
30          return self.pass_dict.copy()
31
32      def add_user(self,user):
33          """Add good user name and strong password to password dictionary."""
34          if not isinstance(user,str) or not user.isalnum():
35              raise NameException
36          if user in self.pass_dict:
37              raise UserException
38          pass_str = input('New password:')
39          # strong password must have digits, uppercase and punctuation
40          if  not (check_pass(pass_str, string.digits) and\
41                      check_pass(pass_str, string.ascii_uppercase) and\
42                      check_pass(pass_str, string.punctuation)):
43              raise PasswordException
44
45      def validate(self,user):
46          """Return True, if valid user and password."""
47          if not isinstance(user,str) or not user.isalnum():
48              raise NameException
49          if user not in self.pass_dict:
50              raise UserException
51          password = input('Passwd:')
52          return self.pass_dict[user]==password
```